04

LINKED LIST

Dr. Mahmoud Alnamoly

Linked List ?

A linked list is a flexible data structure that consists of elements called **nodes**, each containing **data** and a **pointer** to the next node.

Node:

Each element of a linked list is called node. A node has two components:

- ✓ **Data**: The actual value or information that needs to be stored.
- ✓ **Next**: A pointer (or reference) to the next node in the linked list.

Head:

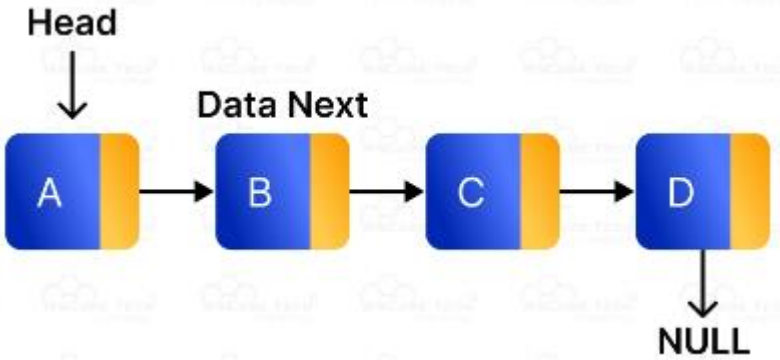
The first node in a linked list is called the head. It is the entry point to the list and used as a reference point to traverse it.

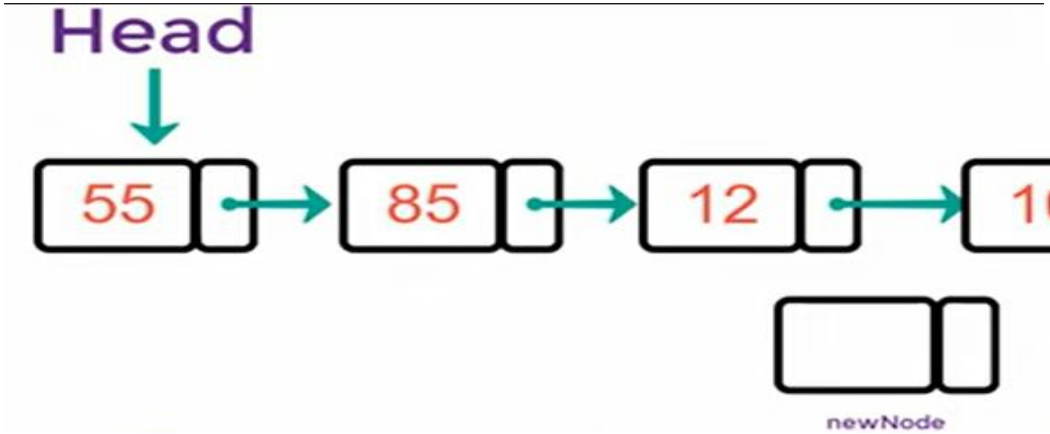
Null:

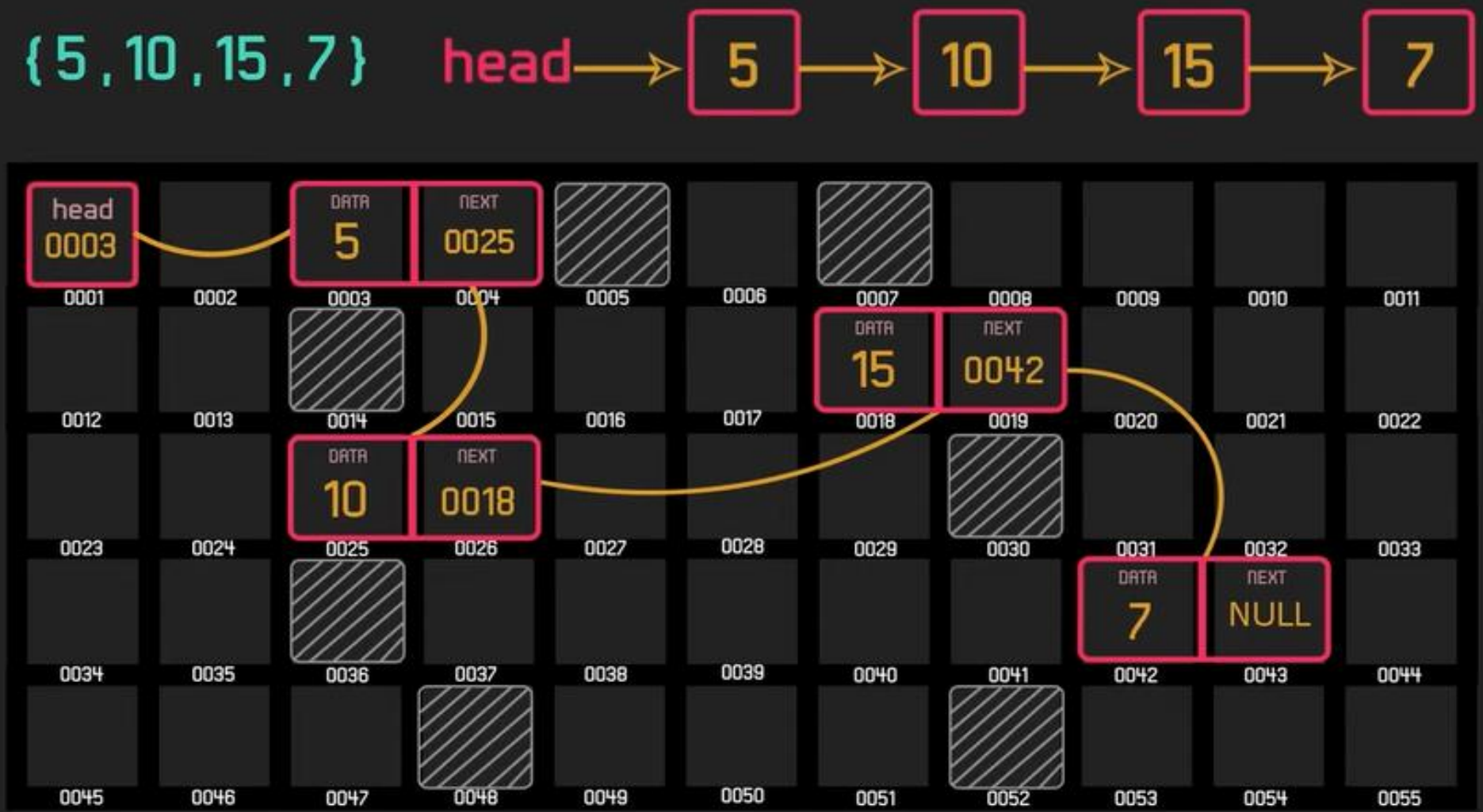
The last node of a linked list, which points to null, indicating the end of the list.

Linked List ?

A linked list is a flexible data structure that consists of elements called **nodes**, each containing **data** and a **reference** to the next node






{5, 10, 15, 7}
head
5
10
15
7
head
0003
DATA
5
NEXT
0025
DATA
15
NEXT
0042
DATA
10
NEXT
0018
DATA
7
NEXT
NULL
0001 0002 0003 0004 0005 0006 0007 0008 0009 0010 0011
0012 0013 0014 0015 0016 0017 0018 0019 0020 0021 0022
0023 0024 0025 0026 0027 0028 0029 0030 0031 0032 0033
0034 0035 0036 0037 0038 0039 0040 0041 0042 0043 0044
0045 0046 0047 0048 0049 0050 0051 0052 0053 0054 0055

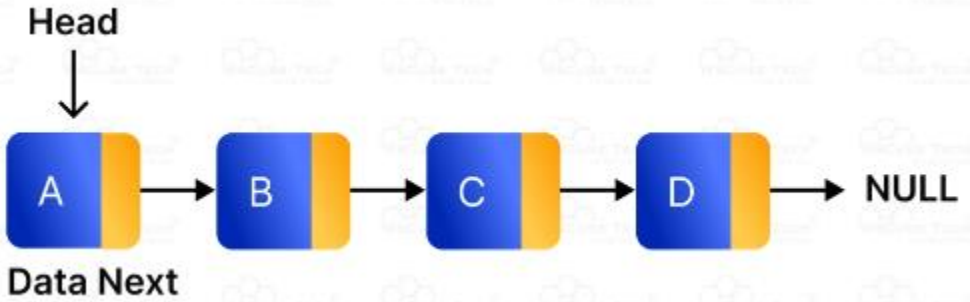
Types of Linked List

There are three primary types of linked lists:-

1. Singly Linked List

A singly linked list is a collection of nodes where each node contains data and a reference to the next node in the sequence. This structure allows for a simple, one-directional traversal from the head to the end of the list.

Operations like adding or removing tasks involve adjusting the 'next' reference of the preceding node, which is straightforward but only allows moving forward through the list.



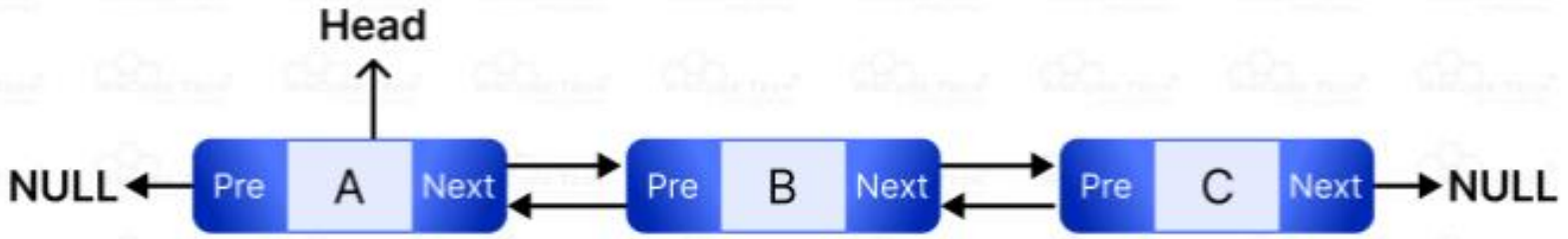
Types of Linked List

There are three primary types of linked lists:-

2. Doubly Linked List

Doubly linked lists expand on singly linked lists by having each node contain two references: one pointing to the next node and one to the previous node. This allows nodes to be traversed in both forward and backward directions.

This setup is particularly useful for applications like image viewers or document viewers where users may need to go forward and backward between images or pages



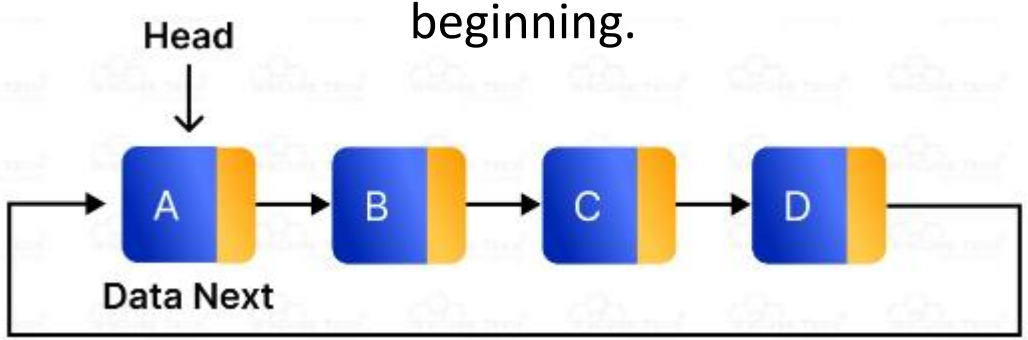
Types of Linked List

There are three primary types of linked lists:-

3. Circular Linked List

A circular linked list is a type of data structure where each node points to the next node, and the last node points back to the first, forming a circle.

This setup allows for an endless cycle of node traversal, which can be particularly useful in applications where the end of the list naturally reconnects to the beginning.

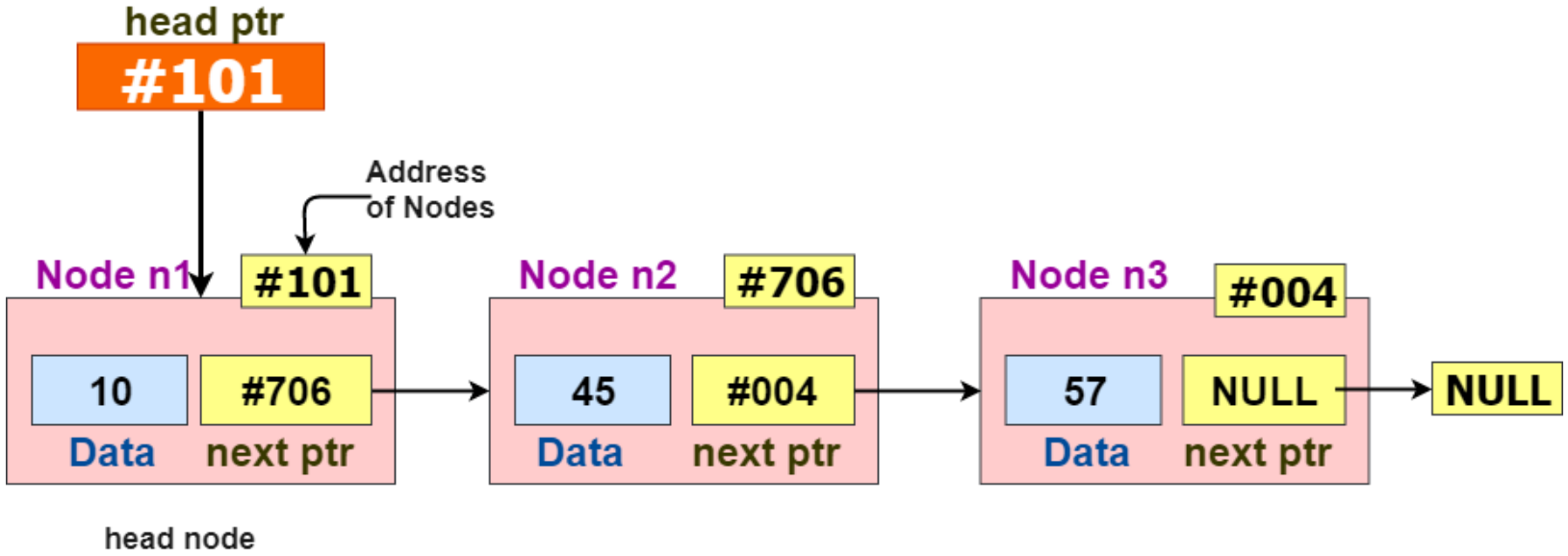


Single Linked List

A singly linked list is a linear data structure where each element (node) is connected to the next one via a **pointer**, forming a sequence.

Each node contains two parts:

Data: The value stored in the node. **Next**: A pointer to the next node in the list.



Single Linked List

Before going on, we need to make a quick revision on **structure**

Structure is a collection of variables of different data types under a single name. It is similar to a class in that, both holds a collection of data of different data types.

For example: You want to store some information about a person: **his/her name**, **citizenship number** and **salary**. You can easily create different variables name, citNo, salary to store these information separately. A better approach will be to have a collection of all related information under a **single name Person**, and use it for every person. Now, the code looks much cleaner, readable and efficient as well. This collection of all related information under a single name Person is a **structure**.

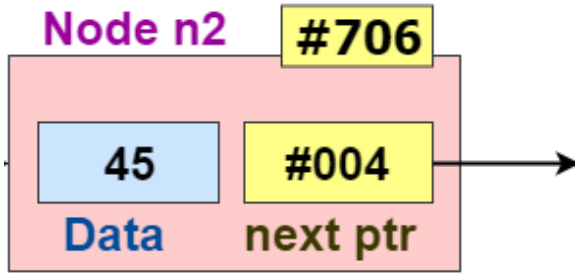
```
struct Person {
    std::string name;
    int age;
    float salary;
};
```

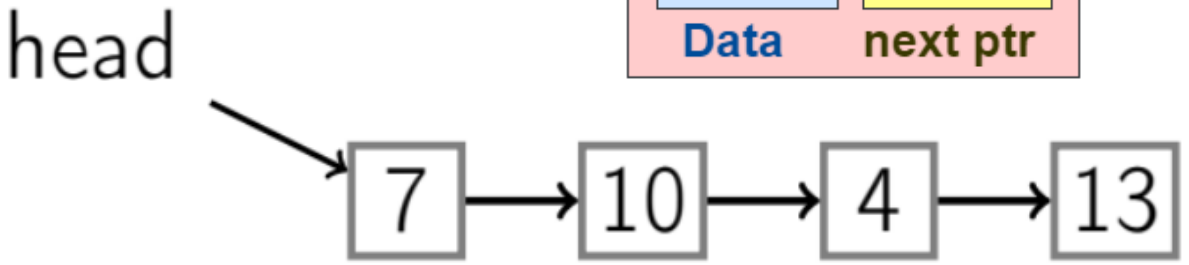
Self Referential Structure

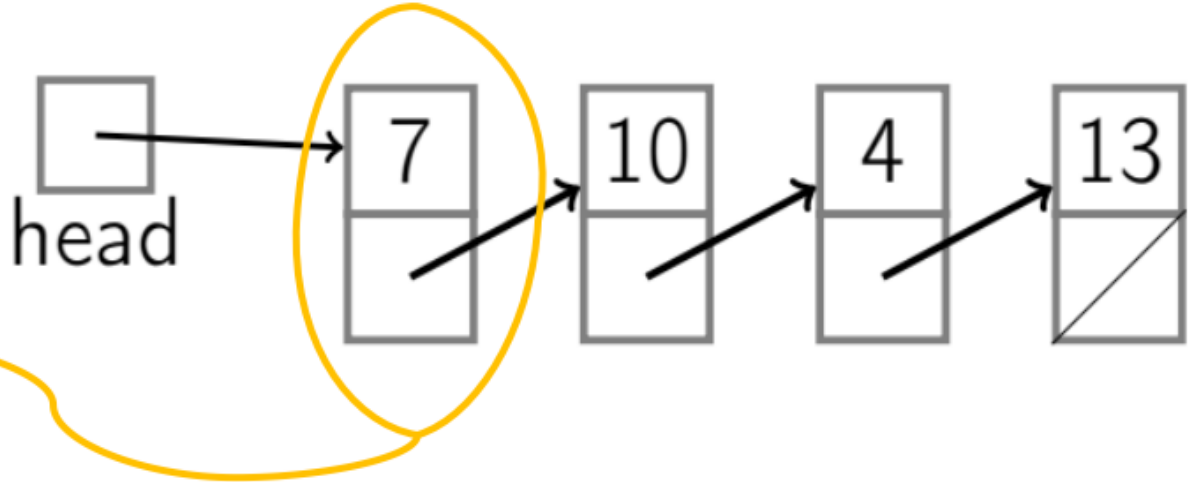
Dr. Mahmoud Alnamoly

How to represent a Node in SLL

- Node contains:
  - key
  - next pointer



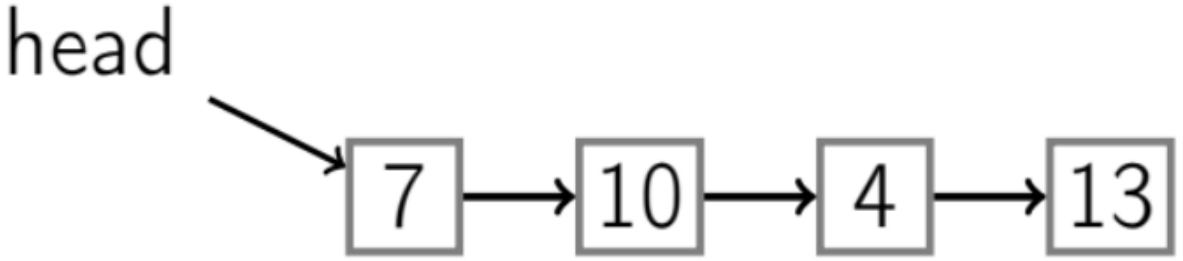


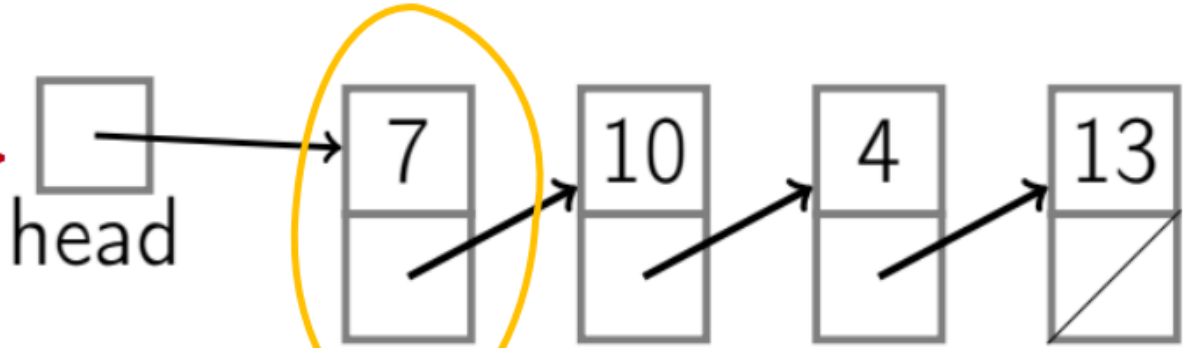



```
struct Node{
    int key;
    Node *next;
};
```

How to represent a Node in SLL

- Node contains:
  - key
  - next pointer

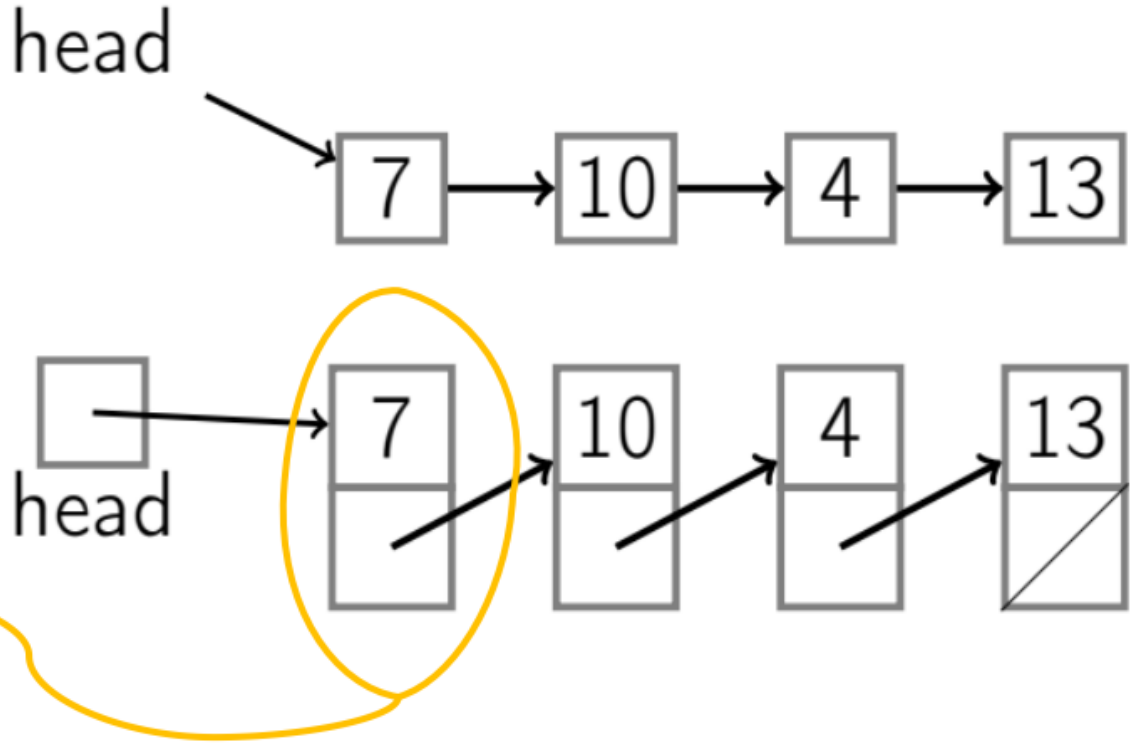




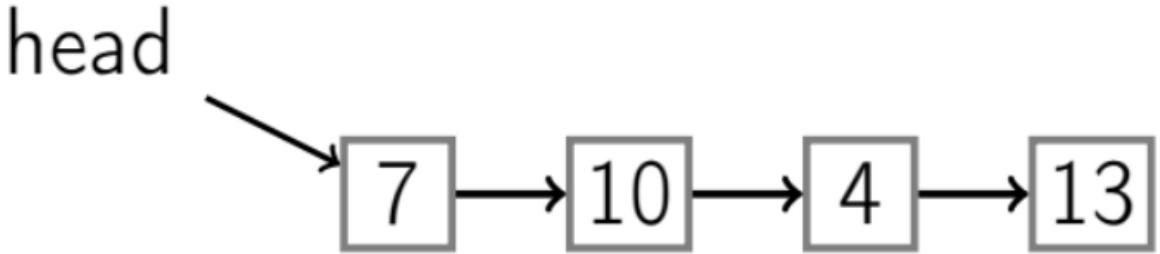

```
template <typename T>
struct Node{
    T key;
    Node *next;
};
```

How to represent a Node in SLL

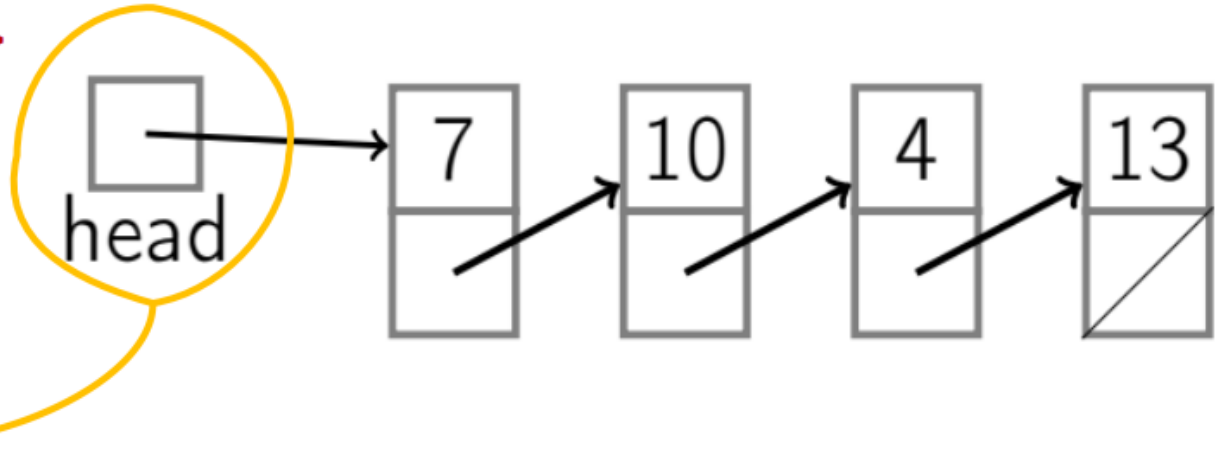
```
template <typename T>
struct Node{
    T key;
    Node *next;
    Node(){
      next = NULL;
    }
    Node(T x){
      key = x;
      next = NULL;
    }
};
```



How to represent a Node in SLL




```
template <typename T>
class SinglyLL{
    private:
        Node *head;
        size_t size;
};
```



How to represent a Node in SLL

```
struct Node {
public:
    int data;         // Data stored in the node
    Node* next;       // Pointer to the next node

    // Constructor
    Node(int value) : data(value), next(nullptr) {}
};


class Node {
public:
    int data;         // Data stored in the node
    Node* next;       // Pointer to the next node

    // Constructor
    Node(int value) : data(value), next(nullptr) {}
};
```

linked list = some of Nodes connected with pointer

```
class LinkedList {
private:
    Node* head;        // Pointer to the head of the list
    Node* tail;        // Pointer to the tail of the list

public:
    // Constructor
    LinkedList() : head(nullptr), tail(nullptr) {}
    ……
    ……
    ……
    // Write here all linked list operations (Functions)
    ……
    ……
    ……
```

Single Linked List operations

Singly Linked list is a type of Linked List Data structure which behaves like a one way list/chain. The reason it is called a one way list or one way chain is because we can only traverse this list in one direction, start from the head node to the end.

Understanding how to perform operations on linked lists is crucial for using them effectively in programming.

- ✓ Access the elements in list (Traverse)
- ✓ Insert a new node in list at (Beginning, End, given location)
- ✓ delete a node from list from (Beginning, End, given location)
- ✓ Find element at specific location (Search)
- ✓ Reverse linked list elements
- ✓ GetSize of list
- ✓ Clear list

Dr. Mahmoud Alnamoly

✓ Traverse

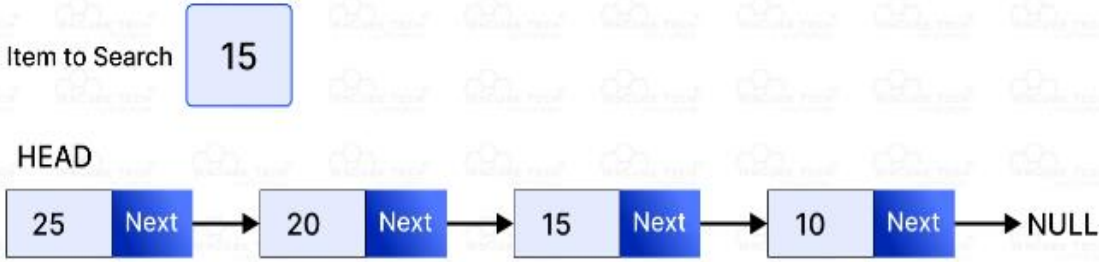
Traversal means going through each node in the list from the beginning to the end to access or modify data.

Steps:

- ✓ Start from the head.
- ✓ Traverse through each node and print its data.

```
// Display the list
void display() const {
    Node* current = head;
    while (current) {
        std::cout << current->data << " ";
        current = current->next; // Move to the next node
    }
    std::cout << std::endl;
}
```

✓ Search



Steps:

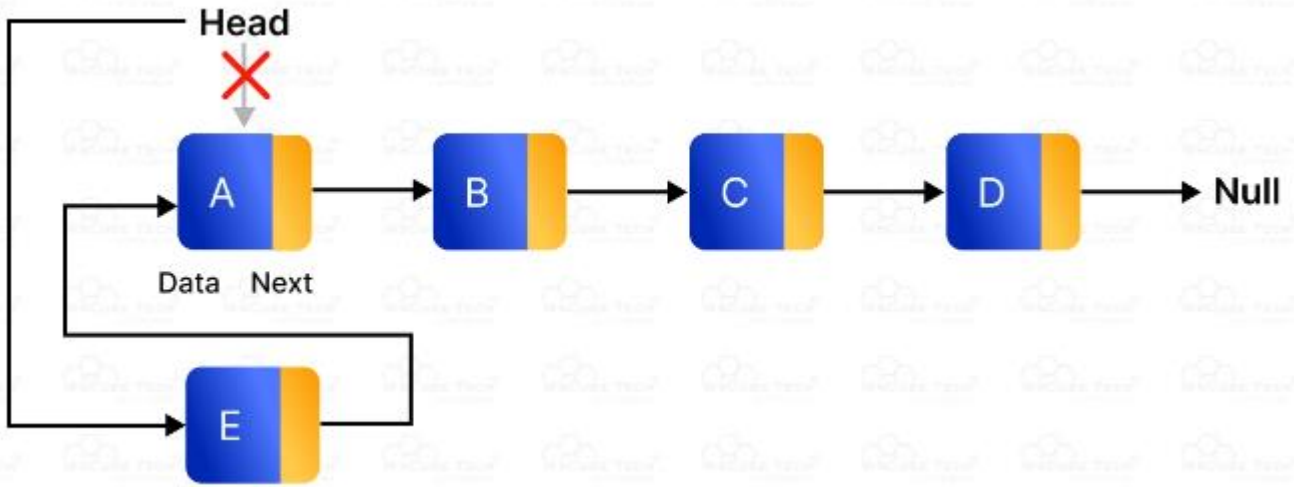
- ✓ Start from the head.
- ✓ Traverse through each node and compare its data with the target value.

```
// Search for an element
bool search(int value) const {
    Node* current = head;
    while (current) {
        if (current->data == value) {
            return true; // Found the value
        }
        current = current->next; // Move to the next node
    }
    return false; // Value not found
}
```

✓ Insert node: Case #1:Insertion at the Beginning

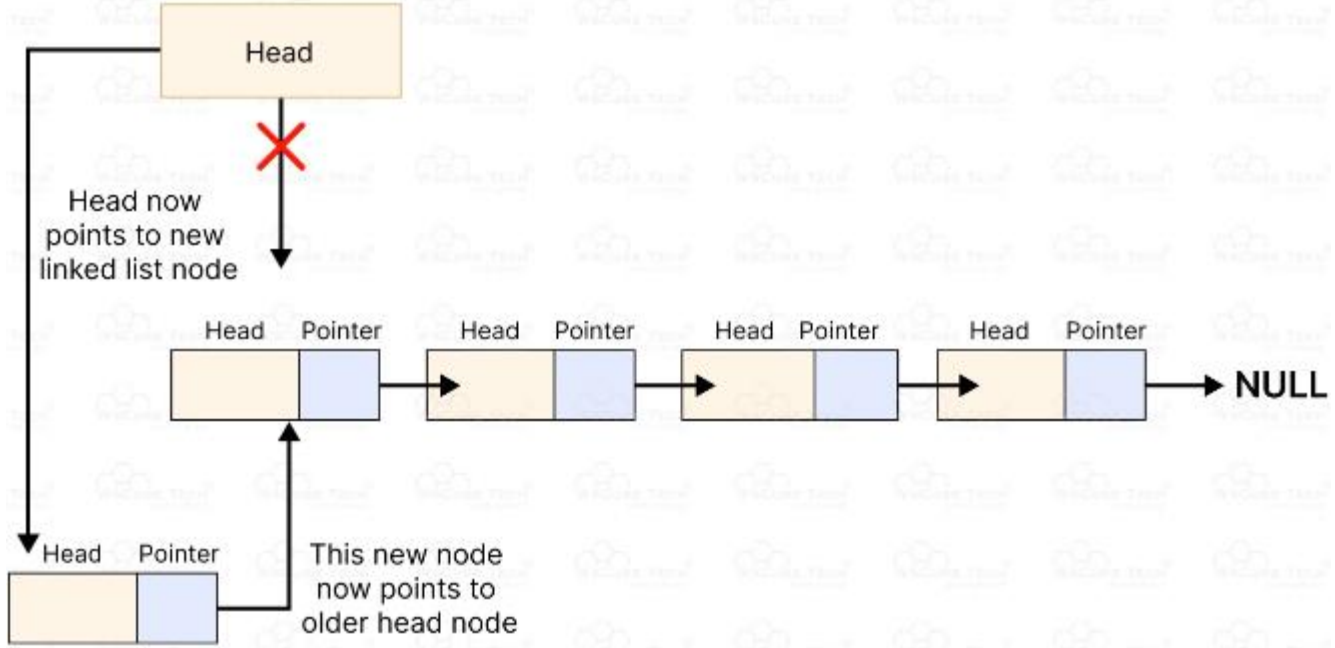
Steps:

- ✓ Create a new node.
- ✓ Set the new node's next pointer to the current head.
- ✓ Update the head to point to the new node.



✓ Insert node: Case #1:Insertion at the Beginning

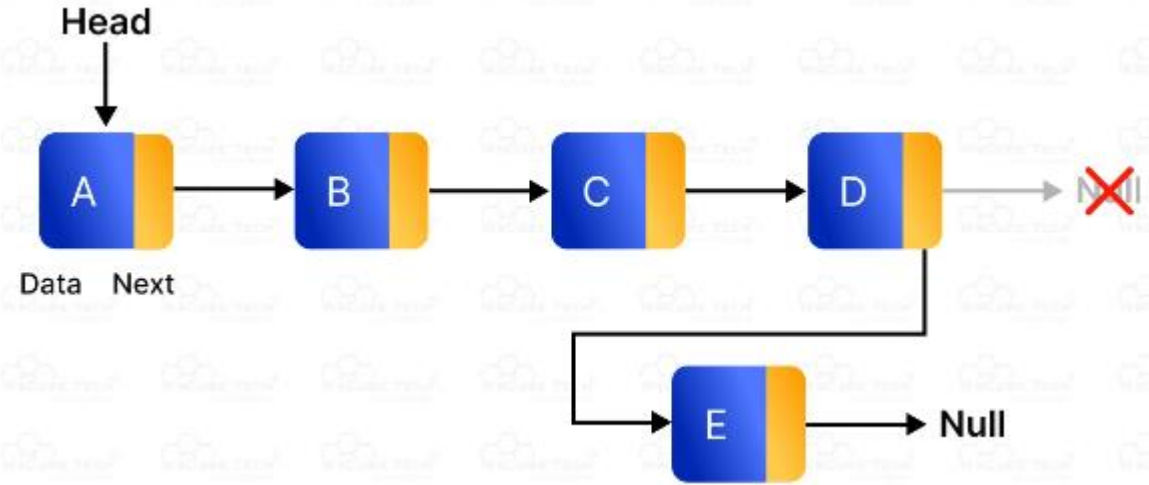
```
// Function to insert a node at the beginning of the list
void insertAtBeginning(int value) {
    Node* newNode = new Node(value); // Create a new node
    newNode->next = head; // Point the new node's next to the current head
    head = newNode; // Update head to point to the new node
}
```



✓ Insert node: Case #2:Insertion at the End

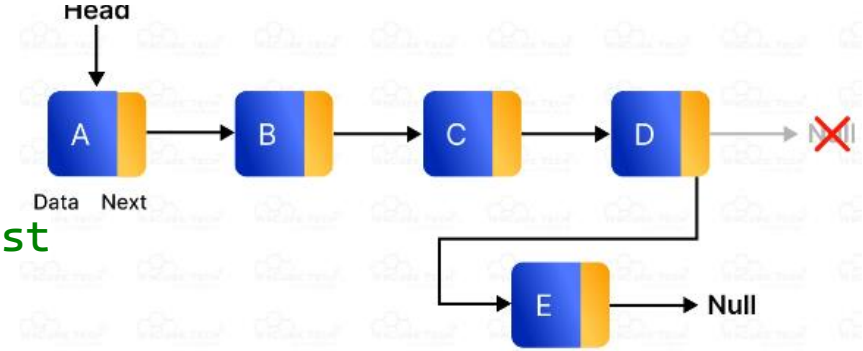
Steps:

- ✓ Create a new node.
- ✓ Traverse to the end of the list.
- ✓ Set the last node's next pointer to the new node.



✓ Insert node: Case #2:Insertion at the End

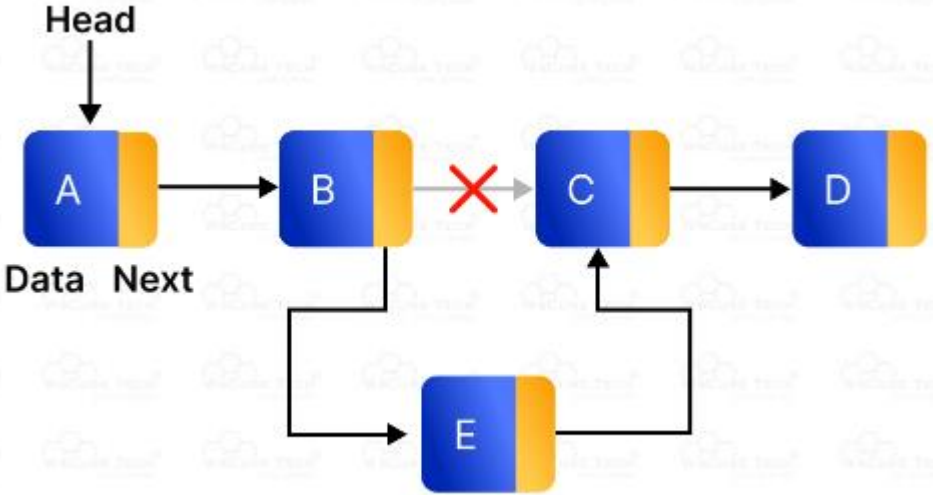
```
// Function to insert a node at the end of the list
void insertAtEnd(int value) {
    Node* newNode = new Node(value);  // Create a new node
    if (head == nullptr) {
        // If list is empty, set head to the new node
        head = newNode;
    }
    else {
        Node* temp = head;
        // Traverse to the end of the list
        while (temp->next != nullptr) {
            temp = temp->next;
        }
        // Set the last node's next to the new node
        temp->next = newNode;
    }
}
```



✓ Insert node: Case #3:Insertion at a Given Position

Steps:

- ✓ Create a new node.
- ✓ Traverse to the node before the desired position.
- ✓ Set the new node's next pointer to the current node's next.
- ✓ Update the current node's next pointer to the new node.



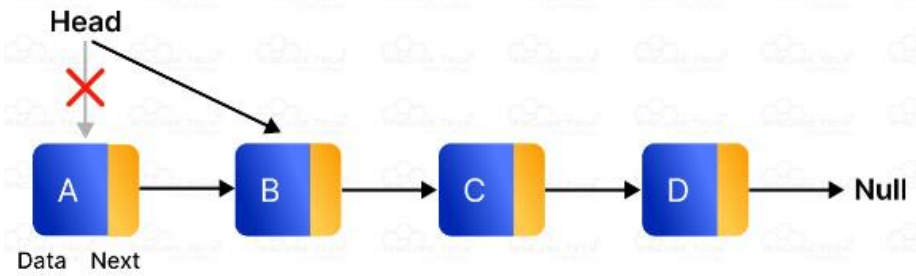
✓ Insert node: Case #3:Insertion at a Given Position

```
// Function to insert a node at a specific position
    void insertAtPosition(int value, int position) {
        Node* newNode = new Node(value);  // Create a new node
        if (position == 1) {
            // Insert at the beginning if position is 1
            newNode->next = head;
            head = newNode; }
        else {
            Node* temp = head;
            // Traverse to the position before the insertion point
            for (int i = 1; i < position - 1 && temp != nullptr; i++) {
                temp = temp->next;
            }
            if (temp == nullptr) {
                std::cout << "Position out of bounds" << std::endl; }
            else {
                newNode->next = temp->next; // Link the new node to the next node
                temp->next = newNode; // Link the previous node to the new node
            }
        }
    }
```

✓ delete node: Case #1:deleting from the beginning

Steps:

- ✓ Check if the list is empty.
- ✓ Set the head to the next node.

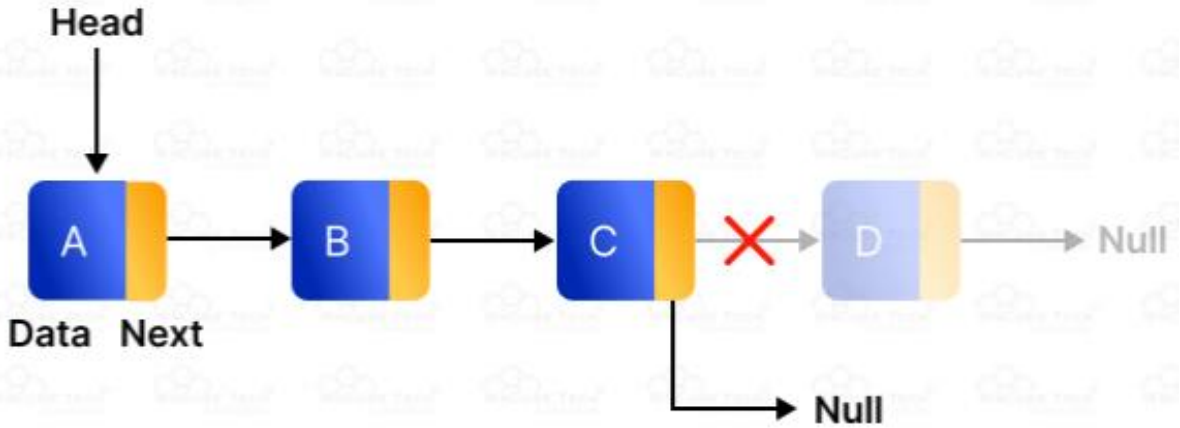



```
// Function to delete a node at the beginning of the list
void deleteAtBeginning() {
    if (head == nullptr) {
        cout << "List is empty, nothing to delete." << endl;
        return;
    }
    Node* temp = head;
    head = head->next;   // Move head to the next node
    delete temp;         // Delete the old head
}
```

✓ delete node: Case #2:deleting at the end

Steps:

- ✓ Check if the list is empty.
- ✓ Traverse to the second-to-last node.
- ✓ Set the second-to-last node's next pointer to None



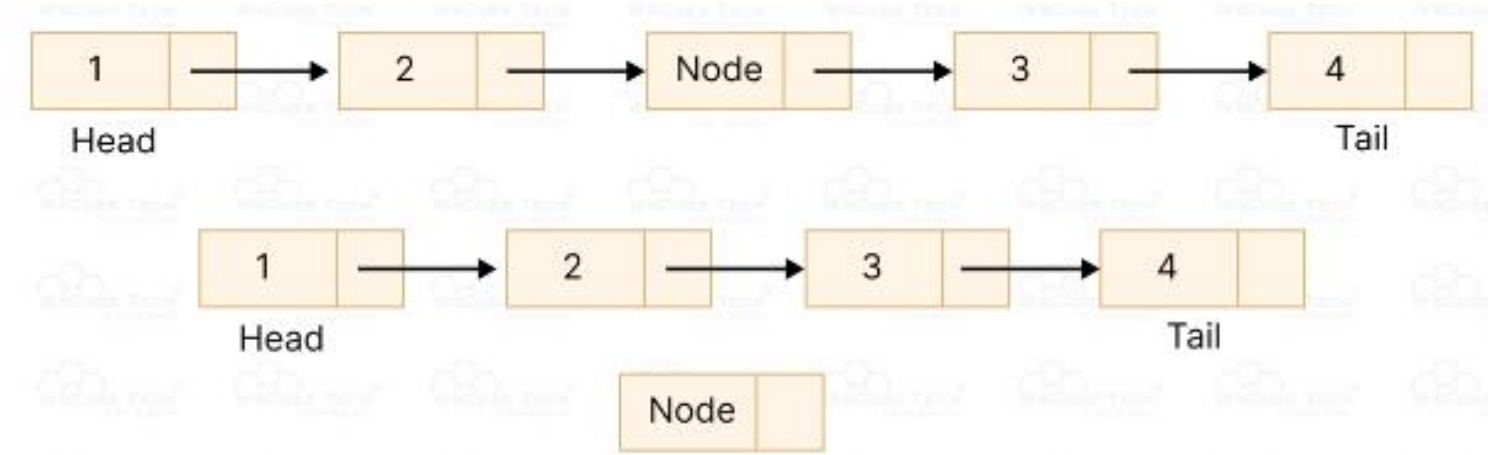
✓ delete node: Case #2:deleting at the end

```
void deleteAtEnd() {
    if (head == nullptr) {
        cout << "List is empty, nothing to delete." << endl;
        return;
    }
    if (head->next == nullptr) {
        delete head;   // If there's only one node
        head = nullptr;
        return;
    }
    Node* temp = head;
    // Traverse to the second last node
    while (temp->next->next != nullptr) {
        temp = temp->next;
    }
    delete temp->next;  // Delete the last node
    temp->next = nullptr;  // Set the new last node's next to nullptr
}
```

✓ delete node: Case #3:deleting from a specific position

Steps:

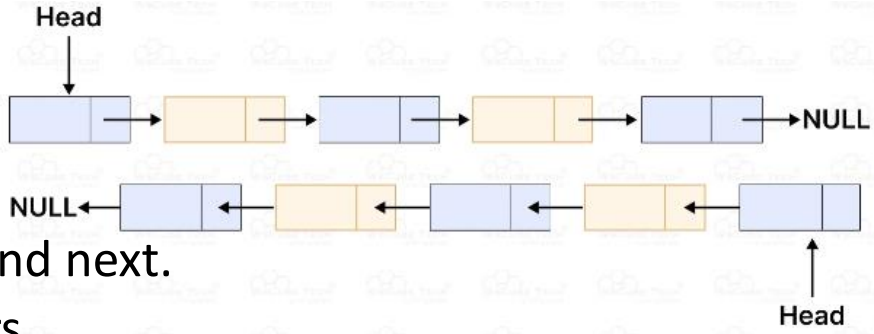
- ✓ Check if the list is empty.
- ✓ Traverse to the node before the desired position.
- ✓ Update the node's next pointer to skip the node at the desired position.



✓ delete node: Case #3:deleting from a specific position

```
void deleteAtPosition(int position) {
    if (position < 1) {
        cout << "Invalid position!" << endl;
        return;}
    if (head == nullptr) {
        cout << "List is empty, nothing to delete." << endl;
        return;}
    if (position == 1) {
        deleteAtBeginning(); // Delete the head node
        return;}
    Node* temp = head;
    // Traverse to the node before the one we want to delete
    for (int i = 1; i < position - 1 && temp->next != nullptr; i++) {
        temp = temp->next;}
    if (temp->next == nullptr) {
        cout << "Position out of bounds!" << endl;}
    else {
        Node* nodeToDelete = temp->next;
        temp->next = nodeToDelete->next;  // Adjust the next pointer
        delete nodeToDelete;              // Delete the node
    }}
```

✓ Reverse



Steps:

- ✓ Initialize three pointers: prev, current, and next.
- ✓ Traverse the list and reverse the pointers

```
void reverse() {
    Node* prev = nullptr;
    Node* current = head;
    tail = head; // Update tail to current head

    while (current) {
        Node* nextNode = current->next; // Store next node
        current->next = prev; // Reverse the link
        prev = current; // Move prev to current
        current = nextNode; // Move to next node
    }
    head = prev; // Update head to the new front
}
```

✓ Length of linked list (Size)

Steps:

- ✓ Initialize a counter.
- ✓ Traverse through each node and increment the counter.

```
// Function to get the length of the linked list
int getLength() const {
    int length = 0;
    Node* temp = head;
    while (temp != nullptr) {
        length++;
        temp = temp->next;
    }
    return length;
}
```

✓ Clear linked list

```
// Clear the list
void clear() {
    while (head) {
        Node* temp = head;
        head = head->next; // Move head to the next node
        delete temp;       // Free the old head
    }
    tail = nullptr; // Reset tail
}
```

Array vs linked list

✓ Both of arrays and linked lists are a **linear** collection of data elements.

Array data structure

- Array data structures store their elements in contiguous memory locations
- It allow random access of data

Linked List data structure

- linked list data structures do not store their elements in contiguous memory locations; instead each node points to the next node through pointer.
- a linked list does not allow random access of data; Nodes in a linked list can be accessed only in a sequential manner

Real-life Applications

Image Viewing Applications

When an image viewer allows you to cycle through photos (previous and next), a doubly linked list can be used to keep track of the images.

Web Browser History

A browser’s history functionality can be implemented using a doubly linked list. Users can navigate back and forth between previously visited web pages.

Music/Media Players

Playlist management: Doubly linked lists are used to navigate forward and backward in playlists. This allows efficient browsing through songs or videos (previous/next functionalities).

Each song in the playlist can be thought of as a node in a linked list, with each node pointing to the next song. Adding or removing songs from your playlist doesn't require shifting all other songs; you simply adjust the links.

Stack

>>> Next Lecture

2nd Project

SinglyLinkedlist_PL

Deadline
28/11/2024

Dr. Mahmoud Alnamoly